03 fév 2009

CMake : la relève dans la construction de projets

Catégorie : Programmation Tags : GLMF

~~Retrouvez cet article dans : Linux Magazine 92~~

La compilation (que nous appellerons plus généralement « construction » dans cet article) de projets logiciels un tant soit peu volumineux fait généralement appel à un outil dédié, chargé de vérifier que la compilation est possible, de la paramétrer et enfin de l'effectuer. Le couple autoconf/automake est l'outil le plus connu permettant d'accomplir cette tâche, mais il souffre de certains défauts, dont sa complexité n'est pas le moindre. Nous allons découvrir CMake, qui se veut une alternative plus simple, plus portable et plus élégante.

En effet, si la plupart des projets libres utilisent le célèbre triplet ~~autoconf/automake/libtool~~ (ou ~~autotools~~) pour gérer leur compilation (le fameux ~~./configure; make; make install~~), l'utilisation de celui-ci ne s'avère pas des plus aisées… et peut rapidement devenir un véritable casse-tête. CMake prétend remplacer ce vénérable système de construction par un seul outil, plus cohérent, plus performant et également plus simple à mettre en œuvre. Le présent article d'introduction à cet outil s'adresse autant aux personnes maîtrisant les autotools qu'à ceux qui n'ont jamais appelé gcc que manuellement. Pour ces derniers, un rappel de la problématique derrière la

construction de projets s'impose.

La construction d'un projet, c'est...

La construction d'un projet désigne toutes les actions permettant de transformer le code source d'un projet vers une forme binaire exécutable et adaptée à un système donné. Elle comprend donc bien évidemment la compilation, mais d'autres actions peuvent intervenir : génération de fichiers de configuration, de la documentation, tests de bon fonctionnement, etc.
Sous Unix/Linux, la construction d'un projet est typiquement réalisée par la commande ~~make~~. Celle-ci lit la description du projet depuis un fichier appelé ~~Makefile~~. Un ~~Makefile~~ est constitué de cibles, de dépendances nécessaires à la construction de ces cibles, et de leurs règles de construction. À partir de ces éléments, ~~make~~ est capable de déterminer, d'une part, les éléments du projet qui nécessitent d'être construits pour obtenir le résultat désiré, et, d'autre part, l'ordre dans lequel ceux-ci doivent l'être. Mieux, une fois relancée, elle ne reconstruit que les parties du projet qui ont été modifiées depuis le dernier appel (ainsi que celles qui en dépendent). ~~make~~ est donc un outil essentiel pour peu que votre projet se compose de plus d'un fichier source. Étant disponible sur de nombreuses plateformes (tous les Unix, mais également Windows, MacOS, etc.) cet outil est pratiquement considéré comme un standard pour construire un projet.
Cependant, si ~~make~~ est un outil extrêmement répandu, d'autres moyens de construire un projet existent, proposés notamment par certains environnements de développement intégrés. Visual Studio de Microsoft dispose ainsi de son propre constructeur, et KDevelop sa propre gestion de projets basée sur ~~make~~. De plus, un projet repose sur des dépendances (bibliothèques, programmes) qui doivent être présentes pour son bon fonctionnement, voire souvent pour que sa compilation aboutisse. Enfin, le compilateur à appeler et les options de compilation ne sont pas forcément les mêmes selon le système. Comment déjà gérer les disparités entre plateformes, notamment concernant l'emplacement des dépendances, le compilateur à utiliser et la manière de l'appeler ? Une solution primitive consiste à demander à l'utilisateur de « remplir les trous » dans un Makefile existant, ou de commenter/décommenter certaines lignes en fonction de sa plate-forme. Une autre solution est de fournir plusieurs Makefile : un par plate-forme supportée. Mais ces solutions n'ont rien de vraiment satisfaisant, car elles ne prennent en compte qu'un nombre limité de cas et demandent une implication trop forte de la part de la personne qui compile le logiciel.
Alors, au-dessus de ~~make~~, s'est greffée une chaîne d'utilitaires permettant de détecter les bons paramètres de compilation et de générer les Makefile adaptés en fonction du système hôte et de sa configuration. La plus connue est la fameuse chaîne des autotools (~~autoconf~~/~~automake~~/~~libtool~~), mais il en existe

d'autres (imake, etc.). Les autotools sont en fait trois outils écrits pour fonctionner ensemble :

- libtool offre une manière portable de créer des bibliothèques ;
- automake permet d'écrire des squelettes de Makefile portables ;
- autoconf permet de décrire les dépendances du projet, de le paramétrer, et génère à partir des squelettes fournis par automake des Makefile capables d'utiliser libtool.

autoconf crée en réalité le fameux script shell configure qui détermine la présence des dépendances et génère les Makefile qui seront utilisés pendant la compilation : configure crée les Makefile, make les utilise pour compiler le projet. Ainsi, l'utilisateur n'a pas à se soucier des spécificités de sa plate-forme, qui sont prises en charge par ces outils. Autre avantage pour l'utilisateur, celui-ci n'a pas besoin de disposer des autotools pour compiler un projet les utilisant, étant donné que tout est géré par le script shell configure. Tout va donc pour le mieux dans le meilleur des mondes ? Pas tout à fait.

...une galère

Mais alors, quel est le problème avec les autotools ?
Tout d'abord, la chaîne des autotools est très loin de former un ensemble cohérent, notamment dans la syntaxe utilisée. Ainsi, autoconf convertit le fichier configure.ac, écrit dans le langage m4, vers le script shell configure ; automake, écrit en Perl, lit une pseudo-syntaxe de Makefile et en produit une autre ; l'utilisation des spécificités de libtool nécessite de maîtriser une syntaxe additionnelle. Rien que pour compiler un projet, 4 nouvelles syntaxes à connaître !
Ensuite, automake semble suivre une règle d'or : celle de rendre chaque nouvelle version incompatible avec les précédentes. Si bien qu'il n'est pas rare d'avoir 2 ou 3 versions d'automake installées sur un même système, pour pouvoir construire différents projets.
Pour terminer, les messages d'erreur de chacun de ces outils sont bien évidemment propres à chacun et cryptiques au possible, si bien que les autotools se sont vus affublés du sobriquet de « autohell ». Ces problèmes ont contribué à qualifier la construction de projets de processus pénible et difficile, accessible uniquement à quelques gourous. Mais en y réfléchissant un peu… est-ce si compliqué que ça de construire un projet ?

CMake

CMake fait figure d'outsider aux autotools pour construire de projet. Il dispose

d'une communauté de développeurs active, enrichie de nombreux utilisateurs. Récemment, un événement important l'a mis sous le feu des projecteurs : le projet KDE l'a adopté avec succès (et soulagement) pour gérer la construction de la future version 4, en remplacement des autotools. Par rapport aux autotools, CMake se veut :

- Plus simple : là où les autotools demandent de connaître plusieurs syntaxes et d'écrire différents types de fichiers, CMake n'utilise qu'un seul type de fichier. La description des cibles est également plus concise.
- Plus pratique : une seule commande (~~cmake~~) remplace les invocations successives de ~~automake~~, ~~autoconf~~ et du script ~~configure~~.
- Plus rapide : l'exécution de CMake surpasse largement en vitesse celle d'un script ~~configure~~.
- Plus user-friendly : l'utilisateur n'a que faire des commandes de compilation de 15 lignes qui remplissent son terminal. Les Makefile générés par CMake informent de la progression de la construction par un message sur la ressource actuellement construite, ainsi qu'un pourcentage de complétion.
- Plus portable : tout comme les autotools, CMake sait bien entendu générer des ~~Makefile~~, mais peut aussi produire des fichiers de projet exploitables par KDevelop et Microsoft Visual Studio.

Beaucoup de promesses donc, mais voyons jusqu'à quel point elles sont tenues au travers d'un petit exemple. Nous travaillerons ici avec la version 2.4 de CMake.

Exemple pratique

Avec les autotools, la configuration de la construction d'un projet se fait au travers de plusieurs fichiers. CMake, au contraire, centralise tout dans un seul fichier, nommé ~~CMakeLists.txt~~. Il contient les directives permettant de s'assurer que toutes les dépendances sont présentes, ainsi que les définitions des bibliothèques et programmes à construire. Nous allons voir comment tout cela fonctionne avec un petit exemple de projet, très basique, définissant une bibliothèque fournissant une implémentation de la fonction ~~ceil~~ (qui retourne l'arrondi supérieur d'un nombre flottant) et d'un programme l'utilisant pour obtenir l'arrondi supérieur d'un nombre passé en argument.

Code du programme

Notre « projet » s'articulera donc autour des fichiers suivants : tout d'abord, la bibliothèque ~~myceil~~, définie dans un sous-répertoire ~~lib~~, comprenant

l'implémentation de ceil définie dans lib/my_ceil.c :

```
#include <math.h>
double my_ceil(double d) {
  if (d - floor(d) != 0.0) d += 1.0;
  return floor(d);
}
```

Ensuite, la définition de l'interface de notre bibliothèque dans lib/my_ceil.h :

```
#ifndef __MY_CEIL_H_
#define __MY_CEIL_H_
extern double my_ceil(double d);
#endif
```

Et enfin, notre programme interactif dans main.c :

CMake n'est pas la seule alternative aux Autotools. Le projet Scons est également un candidat intéressant.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include «lib/my_ceil.h»
int main(int argc, char * argv[]) {
  double d = atof(argv[1]);
  printf(«%f\n», my_ceil(d));
  return EXIT_SUCCESS;
}
```

Une fois compilé et lié avec la bibliothèque myceil, ce programme s'invoque en lui passant un nombre flottant en argument.
Tout ceci est bien évidemment très basique (il n'y a ainsi aucun contrôle d'erreur), mais nous étudions l'outil de construction du programme, et non pas le programme en lui-même.

Le fichier CMakeLists.txt

Il nous faut maintenant définir une série de fichiers CMakeLists.txt nous permettant de compiler la bibliothèque myceil, puis le programme ceil et de lier ce dernier avec notre bibliothèque. Un changement dans le code source doit déclencher la recompilation de tous les éléments dont il dépend. Pour corser le jeu, nous allons en plus exiger que le programme soit obligatoirement lié avec la zlib, et avec les bibliothèques X11 si celles-ci sont disponibles. Ces dépendances ne sont bien sûr pas indispensables pour notre projet, mais nous permettront d'étudier d'autres fonctionnalités de CMake.
Voici à quoi ressemble notre fichier CMakeLists.txt en racine du projet :

```
PROJECT(ceil)
```

En premier lieu, il faut donner un nom à notre projet. Dans notre cas, il s'agira de « ceil ». Notez que la syntaxe des commandes CMake n'est pas sensible à la casse. En revanche, les noms de variables le sont.

La ligne suivante s'assure de la présence de la zlib et fait échouer la configuration du projet si celle-ci n'est pas présente grâce au mot-clé ~~REQUIRED~~ :

```
FIND_PACKAGE(ZLIB REQUIRED)
```

Même chose pour les bibliothèques X11, mais celles-ci sont optionnelles :

```
FIND_PACKAGE(X11)
```

Comme nous utilisons la ~~zlib~~, le chemin de ses en-têtes doit être fourni au compilateur. La ligne suivante permet d'obtenir cet effet :

```
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
```

Notez comment la variable ~~ZLIB_INCLUDE_DIR~~, définie par ~~FIND_PACKAGE~~, est utilisée.
Concernant les bibliothèques X11, il y a deux possibilités : soit elles sont présentes sur le système, et, dans ce cas, nous voulons également que le compilateur ait accès aux en-têtes, soit elles ne le sont pas et il n'y a rien à faire. Dans les deux cas, nous allons afficher un petit message pour informer l'utilisateur de leur présence ou pas, ce qui nous permettra d'étudier la syntaxe quelque peu inhabituelle du ~~IF~~ de CMake :

```
IF(X11_FOUND)
  MESSAGE(STATUS “X11 présent»)
  INCLUDE_DIRECTORIES(${X11_INCLUDE_DIR})
ELSE(X11_FOUND)
  MESSAGE(STATUS «X11 absent»)
ENDIF(X11_FOUND)
```

Voilà pour nos dépendances. Nous pouvons maintenant définir notre projet. Comme nous l'avons déjà dit, il se compose d'une bibliothèque partagée (qui s'appellera ~~myceil~~ et se situe dans le sous-répertoire ~~lib~~) et d'un exécutable ~~ceil~~.

La compilation de la bibliothèque s'effectuant dans un sous-répertoire, il nous faudra définir un autre fichier ~~CMakeLists.txt~~ à l'intérieur de ce sous-répertoire, et demander à CMake de le parcourir. Pour cette dernière opération, il nous suffit d'appeler la commande ~~ADD_SUBDIRECTORY~~ dans notre ~~CMakeLists.txt~~ racine :

```
ADD_SUBDIRECTORY(lib)
```

Elle demande à CMake d'évaluer le répertoire passé en paramètre et de prendre en compte toutes ses directives. Nous allons tout de suite écrire le contenu du fichier lib/CMakeLists.txt, qui ne fera que définir les règles de construction de notre bibliothèque.

Commençons par définir une variable contenant les sources de la bibliothèque myceil. Pour cela, nous utilisons la commande SET. Son premier argument est le nom de la variable, les suivants constituent sa valeur :

```
SET(myceil_lib_src my_ceil.c)
```

Si nous avions voulu spécifier plusieurs fichiers sources, nous les aurions ajoutés après le premier en utilisant un espace comme séparateur.

La définition de la bibliothèque partagée myceil se fait ensuite naturellement de la façon suivante, en utilisant la variable que nous venons de déclarer :

```
ADD_LIBRARY(myceil SHARED ${myceil_lib_src})
```

Et voilà, en deux lignes les règles de construction de notre bibliothèque sont définies. Si nous avions voulu produire une bibliothèque statique, nous aurions utilisé le mot clé STATIC en lieu et place de SHARED. Maintenant, retour à notre CMakeLists.txt racine pour définir les règles de construction de notre exécutable ceil. Nous pouvons également passer par une variable pour définir ses fichiers sources, et utiliser la commande ADD_EXECUTABLE de la même manière que nous avons utilisé ADD_LIBRARY :

```
SET(ceil_src main.c)
ADD_EXECUTABLE(ceil ${ceil_src})
```

Une dernière chose : ceil doit être lié avec notre bibliothèque myceil, la bibliothèque mathématique, la zlib, et les bibliothèques X11 si ces dernières sont présentes. La commande TARGET_LINK_LIBRARIES permet de spécifier avec quels autres éléments un exécutable ou une bibliothèque doit être lié :

```
TARGET_LINK_LIBRARIES(ceil myceil m ${ZLIB_LIBRARY} ${X11_LIBRARIES})
```

Et c'est tout ! Si les bibliothèques X11 ne sont pas présentes, la variable X11_LIBRARIES sera vide, et n'aura donc aucun effet sur l'édition des liens.

Construction du projet

Nous pouvons maintenant essayer tout cela. Dans un répertoire, vous aurez créé les fichiers nécessaires :

```
$ ls
CMakeLists.txt  lib/  main.c
$ ls lib
CMakeLists.txt  my_ceil.c  my_ceil.h
```

Lancez la commande ~~cmake~~ . pour générer le ~~Makefile~~ correspondant à notre projet :

```
$ cmake .
-- Check for working C compiler: gcc
-- Check for working C compiler: gcc — works
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: c++
-- Check for working CXX compiler: c++ -- works
-- Found ZLIB: /usr/lib/libz.so
-- Looking for XOpenDisplay in /usr/lib/libX11.so;/usr/lib/libXext.so
-- Looking for XOpenDisplay in /usr/lib/libX11.so;/usr/lib/libXext.so — found
-- Looking for gethostbyname
-- Looking for gethostbyname — found
-- Looking for connect
-- Looking for connect — found
-- Looking for remove
-- Looking for remove — found
-- Looking for shmat
-- Looking for shmat — found
-- Looking for IceConnectionNumber in ICE
-- Looking for IceConnectionNumber in ICE — found
-- X11 présent
-- Configuring done
-- Generating done
-- Build files have been written to: /home/me/CMake/exemple
```

Vous aurez remarqué que tous les tests ont été effectués très rapidement. Notez également l'affichage de notre petit message indiquant que les bibliothèques X11 ont été trouvées. Si ce n'est pas le cas chez vous, ce n'est pas un problème pour la suite de la construction du projet.

Outre un ~~Makefile~~, l'invocation de CMake a créé d'autres fichiers à son usage. Peu importe, nous pouvons maintenant compiler notre projet avec ~~make~~ :

```
$ make
Scanning dependencies of target myceil
[ 50%] Building C object lib/CMakeFiles/myceil.dir/my_ceil.o
Linking C shared library libmyceil.so
[ 50%] Built target myceil
Scanning dependencies of target ceil
[100%] Building C object CmakeFiles/ceil.dir/main.o
Linking C executable ceil
[100%] Built target ceil
```

Vous pouvez maintenant essayer votre programme :

```
$ ./ceil 1.2
2.000000
```

Les habitués aux autotools seront surpris par la concision du rapport de compilation : juste un pourcentage de progression et une description de l'action courante. Est-il besoin de plus ? Parfois oui, pour déboguer les problèmes de compilation. Dans ce cas, il suffit de déclarer la variable d'environnement VERBOSE à 1 avant de lancer make pour obtenir les commandes de construction exécutées :

```
$ VERBOSE=1 make
```

Les classiques make clean et make install sont également de la partie. Pour avoir une liste des cibles définies dans votre Makefile, vous pouvez invoquer make help.

Et comme nous l'avons dit, CMake sait générer autre chose que des Makefile. Vous êtes fan de Kdevelop ? Invoquez CMake avec l'option -G et les fichiers de projet seront générés pour vous !

Paramétrage de la compilation

Nous avons fait une invocation simple et rapide de CMake pour générer les Makefile de notre projet. Cette invocation convient pour compiler notre programme et le lancer, mais ce cas d'utilisation est loin d'être unique.

La sortie générée par CMake peut être paramétrée par de nombreuses variables d'environnement. Celles-ci sont définies selon le même modèle que les définitions de GCC, au travers de l'option '-DNOM_VARIABLE=VALEUR'. Parmi les options les plus fréquentes, on peut lister :

- CMAKE_BUILD_TYPE peut prendre les valeurs Debug (produit un binaire contenant les symboles de debug) ou Release (produit un binaire optimisé et sans symboles de debug).
- CMAKE_INSTALL_PREFIX pointe vers le chemin d'installation du projet (par défaut /usr/local).

Vous avez également la possibilité de définir des paramètres de compilation propres à votre projet. Dans le fichier CMakeLists.txt, la commande OPTION permet de définir des options de type booléen :

```
OPTION(WITH_GUI «Compiler l'interface graphique» OFF)
```

Le premier paramètre est le nom de la variable définissant l'option, le second la description de cette option, et le troisième la valeur par défaut. Plus loin dans le fichier, il est possible de voir si l'option a été activée en testant la

valeur de la variable déclarée :

```
IF(WITH_GUI)
  MESSAGE(STATUS “Compilation de l’interface graphique activée»)
ENDIF(WITH_GUI)
```

D’une manière plus générale, vous pouvez vérifier la présence et la valeur de n’importe quelle variable arbitraire qui aurait été passée à CMake. Par exemple, si nous voulons pouvoir donner le chemin d’un répertoire de données à notre projet tout en assurant une valeur par défaut, nous pouvons utiliser les lignes suivantes :

```
IF (NOT DATA_DIR)
  SET(DATA_DIR «/usr/share/mydatadir»)
ENDIF(NOT DATA_DIR)
MESSAGE(STATUS “Données situées dans ${DATA_DIR}»)
```

Enrichi de ce code, notre projet peut maintenant se paramétrer avec les deux variables nouvellement déclarées :

```
$ cmake -DWITH_GUI=ON -DDATA_DIR=/home/me/datadir .
```

Passage des paramètres de compilation au projet

Il peut s’avérer nécessaire de conditionner la compilation de certaines parties d’un fichier source en fonction des paramètres de compilation. Par exemple, dans un des sources de notre projet, nous pourrions vouloir ne compiler du code que si l’interface graphique doit être compilée, ou si les bibliothèques X11 sont présentes. Il nous faudrait un fichier d’en-tête qui reflète la partie utile du paramétrage de la construction.

C’est à cela que sert la commande ~~CONFIGURE_FILE~~. Elle prend en paramètres un fichier de modèle, ainsi que la destination du traitement de ce fichier. Voyons comment elle fonctionne par l’exemple. Dans notre projet, nous définissons un fichier ~~config.h.cmake~~ dont le contenu est le suivant :

```
#ifndef CONFIG_H__
#define CONFIG_H__
#cmakedefine WITH_GUI 1
#cmakedefine DATA_DIR «${DATA_DIR}»
#cmakedefine X11_FOUND 1
#endif
```

Il suffit ensuite de rajouter la ligne suivante dans le fichier ~~CMakeLists.txt~~ :

```
CONFIGURE_FILE(${CMAKE_SOURCE_DIR}/config.h.cmake ${CMAKE_BINARY_DIR}/config.h)
INCLUDE_DIRECTORIES(${CMAKE_BINARY_DIR})
```

Et lors de l'invocation à CMake, le fichier config.h contenant nos déclarations utiles est créé et peut être inclus dans n'importe lequel de nos sources :

```
#ifndef CONFIG_H__
#define CONFIG_H__
#define WITH_GUI 1
#define DATA_DIR «/usr/share/mydatadir»
#define X11_FOUND 1
#endif
```

Pourquoi une nouvelle invocation à INCLUDE_DIRECTORIES après l'appel à CONFIGURE_FILE ? Le fichier config.h.cmake fait partie de notre projet et est logiquement situé dans son répertoire source. Par contre, le fichier config.h généré est spécifique à notre compilation et doit être placé dans le répertoire de construction… qui n'est pas forcément le même que le répertoire des sources et doit donc être indiqué au compilateur si celui-ci veut y trouver des fichiers d'en-tête. Il est en effet possible de séparer répertoire source et répertoire de construction.

Construction séparée

CMake permet sans problème de compiler un projet en dehors de son répertoire source. Pour compiler notre programme précédent, vous auriez pu vous placer dans un répertoire vide et invoquer cmake de la façon suivante :

```
$ cmake /chemin/vers/mon/projet
```

Il suffit ensuite d'appeler make pour compiler le projet sans que le répertoire des sources ne soit affecté. Ceci est particulièrement utile si vous voulez produire différentes versions binaires d'un même projet (une version « debug » et une version « release », par exemple). Il s'agit en fait d'une bonne pratique générale.

Installation du projet construit

Tel que nous avons décrit notre projet, la commande make install n'installera strictement rien. Il nous faut définir dans les fichiers CMakeLists.txt ce que nous désirons installer, et où. La commande INSTALL sert précisément à cela.

```
INSTALL(TYPE fichiers a installer DESTINATION destination)
```

TYPE renseigne sur le type de fichier à installer. Parmi les valeurs possibles, TARGETS indique que nous souhaitons installer des cibles construites par CMake, et FILES sert pour installer des fichiers réguliers. Vient ensuite la liste des cibles ou fichiers à installer, le mot clé DESTINATION, et finalement la

destination proprement dite. Si cette dernière désigne un chemin relatif, celui-ci le sera par rapport au préfixe d'installation (défini par la variable CMAKE_INSTALL_PREFIX).

Voici comment nous pouvons compléter notre projet ceil avec ses cibles d'installation. Dans CMakeLists.txt, nous ajoutons une ligne pour installer le programme ceil dans le répertoire bin du préfixe d'installation :

```
INSTALL(TARGETS ceil DESTINATION «bin»)
```

Et dans lib/CMakeLists.txt, nous ajoutons deux cibles : une pour installer le fichier d'en-tête (qui n'est pas généré par CMake) dans include, et une autre pour installer la bibliothèque construite par CMake dans lib.

```
INSTALL(FILES my_ceil.h DESTINATION «include»)
INSTALL(TARGETS myceil DESTINATION "lib")
```

La commande make install donne maintenant le résultat attendu !

Conclusion

Cet article ne se veut en rien exhaustif : il avait pour but de démontrer l'obsolescence des autotools qui, après avoir bravement rempli leur mission pendant de nombreuses années, ont bien mérité une mise à la retraite. Nous n'avons effectué qu'une introduction rapide à CMake afin de montrer sa simplicité et sa logique, mais CMake est en réalité un outil très puissant et personnalisable qui peut également gérer les tests unitaires (avec ctest) et le packaging des projets (avec cpack). Vous pouvez trouver plus d'informations à ces sujets sur son site officiel.

Gageons et espérons que l'adoption de ce nouvel outil de construction par des projets aussi connus que KDE contribue à son utilisation quotidienne, ce qui aura sans nul doute comme effet de rendre la construction de projets plus simple, plus sûre, plus portable et surtout… moins effrayante.

Liens

- Site officiel de CMake : http://www.cmake.org
- Le code complet de l'exemple de l'article : http://www.gnurou.org/documents/linuxmag/cmake/exemple.tar.gz

Retrouvez cet article dans : Linux Magazine 92

Posté par (La rédaction) | Signature : Alexandre Courbot | Article paru dans

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

« Précédent Aller au contenu »
Identifiez-vous
Inscription
S'abonner à UNIX Garden

Articles de 1ère page

Catégories

- Administration réseau
- Administration système
- Agenda-Interview
- Audio-vidéo
- Bureautique
- Comprendre
- Distribution
- Embarqué
- Environnement de bureau
- Graphisme
- Jeux
- Matériel
- News
- Programmation

- Réfléchir
- Sécurité
- Utilitaires
- Web

Articles secondaires

- 30/10/2008
  Google Gears : les services de Google offline

  Lancé à l'occasion du Google Developer Day 2007 (le 31 mai dernier), Google Gears est une extension open source pour Firefox et Internet Explorer permettant de continuer à accéder à des services et applications Google, même si l'on est déconnecté.…

  Voir l'article...
  7/8/2008
  Trois questions à...

  Alexis Nikichine, développeur chez IDM, la société qui a conçu l'interface et le moteur de recherche de l'EHM.…

  Voir l'article...
  11/7/2008
  Protéger une page avec un mot de passe

  En général, le problème n'est pas de protéger une page, mais de protéger le répertoire qui la contient. Avec Apache, vous pouvez mettre un fichier ~~.htaccess~~ dans le répertoire à protéger.…

  Voir l'article...
  6/7/2008
  hypermail : Conversion mbox vers HTML

  Comment conserver tous vos échanges de mails, ou du moins, tous vos mails reçus depuis des années ? mbox, maildir, texte… les formats ne manquent pas. …

  Voir l'article...
  6/7/2008
  iozone3 : Benchmark de disque

  En fonction de l'utilisation de votre système, et dans bien des cas, les performances des disques et des systèmes de fichiers sont très importantes.…

Voir l'article...
1/7/2008
Augmentez le trafic sur votre blog !

Google Blog Search (http://blogsearch.google.fr/) est un moteur de recherche consacré aux blogs, l'un des nombreux services proposés par la célèbre firme californienne....

Voir l'article...

- **GNU/Linux Magazine**

  - GNU/Linux Magazine N°113 - Février 2009 - Chez votre marchand de journaux
  - Édito : GNU/Linux Magazine 113
  - Un petit sondage pour améliorer nos magazines
  - GNU/Linux Magazine HS N°40 - Janvier/Février 2009 - Chez votre marchand de journaux
  - Edito : GNU/Linux Magazine HS 40

- **GNU/Linux Pratique**

  - Linux Pratique HS N°17 - Mars/Avril 2009 - Chez votre marchand de journaux
  - Édito : Linux Pratique HS N°17
  - Linux Pratique HS 17 - Communiqué de presse
  - Linux Pratique Essentiel N°6 - Février/Mars 2009 - Chez votre marchand de journaux
  - Édito : Linux Pratique Essentiel N°6

- **MISC Magazine**

  - Un petit sondage pour améliorer nos magazines
  - MISC N°41 : La cybercriminalité ...ou quand le net se met au crime organisé - Janvier/Février 2009 - Chez votre marchand de journaux
  - Édito : Misc 41
  - MISC 41 - Communiqué de presse
  - Les Éditions Diamond adhèrent à l'APRIL !